Paradigmas de Programación

Programación orientada a objetos

2do cuatrimestre de 2024

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Distintos modos de concebir la programación

Los paradigmas que hemos visto se fundamentan en la lógica:

Programación funcional	los programas son demostraciones.
Programación lógica	los programas son fórmulas.

Programación orientada a objetos

Se inspira en otras varias disciplinas. Por ejemplo:

- Biología: adaptabilidad y resiliencia de los organismos vivos.
- Arquitectura: diseño de "catedrales en lugar de pirámides".

Los programas están conformados por componentes, llamadas **objetos**, que interactúan intercambiando **mensajes**.

Las entidades físicas o conceptuales del dominio del problema que se quiere modelar se representan como objetos.

El comportamiento de los objetos debe reflejar fielmente aquellos aspectos que nos interesan de las entidades de la "realidad".

Breve perspectiva histórica

La POO surgió alrededor de 1970 para abstraer técnicas comunes de la programación procedural:

1. Pasaje de registros[1] para permitir código reentrante.
   (Alternativa superadora a las variables globales).
2. Agrupamiento de las funciones en módulos.
3. Polimorfismo por indirección:
   un registro contiene punteros a funciones que lo manipulan.

Algunos lenguajes orientados a objetos muy influyentes:

- Simula ..... Dahl & Nygaard, ∼1967
- Smalltalk ..... Kay, ∼1972
- Self ..... Ungar & Smith, ∼1987

[1] Es decir, *tuplas* o `structs`.

Introducción

Conceptos fundamentales de POO

Introducción a la POO en Smalltalk

Objetos y mensajes

1. Un entorno OO está compuesto por **objetos**.
2. Un objeto puede enviar un **mensaje** a otro. Un mensaje representa una solicitud al objeto receptor, para que lleve a cabo una de sus operaciones.
3. La **interfaz** de un objeto es el conjunto de mensajes que es capaz de responder.
4. Un **método** es el procedimiento que usa un objeto para responder un mensaje. Es decir, es la implementación efectiva de la operación solicitada por el mensaje.
5. La forma en la que un objeto lleva a cabo una operación puede depender de **colaboradores externos**[2] así como de su **estado** interno, dado por un conjunto de **colaboradores internos**[3].

[2] También llamados *parámetros* o *argumentos* del mensaje recibido.
[3] También llamados *atributos* o *variables de instancia* del objeto receptor.

Objetos y mensajes

Ejemplo

Objeto que representa un rectángulo de 5×2:

Interfaz:	**área**
Atributos:	`ancho` . . . . . ⟨objeto que representa al 5⟩ `alto` . . . . . ⟨objeto que representa al 2⟩
Métodos:	**área** `^ ancho * alto`

Encapsulamiento

Principio de encapsulamiento

Sólo se puede interactuar con un objeto a través de su interfaz.
El estado interno de un objeto es inaccesible desde el exterior.

Consecuencias del encapsulamiento

1. Alternar entre dos representaciones de una misma entidad no modifica el comportamiento observable del sistema.
   Ejemplo. Un conjunto de enteros se puede representar con una lista enlazada o con un árbol binario balanceado, sin que el usuario pueda notar una diferencia de comportamiento.
2. *"Duck typing".* Un objeto se puede intercambiar por otro que implemente la misma interfaz.
   Ejemplo. Si espero interactuar con un buscador que responde el mensaje "`buscar: texto`", me pueden suministrar un objeto que responde ese mensaje y además mantiene estadísticas de uso.

Variantes de la orientación a objetos

Los entornos OO pueden tener distintas características:

- Envío de mensajes sincrónico vs. asincrónico.
- Envío de mensajes con respuesta vs. sin respuesta.
- Objetos mutables vs. inmutables.
- Clasificación vs. prototipado.
- Herencia simple vs. herencia múltiple.
- ...

El entorno que usaremos (Smalltalk) tiene características típicas.

Clases e instancias

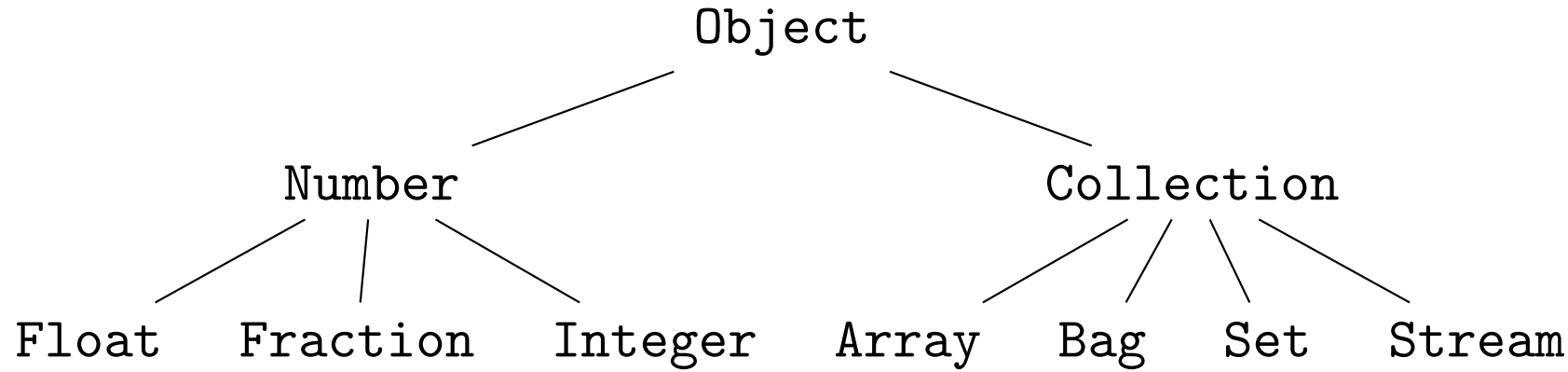
Todo objeto es **instancia** de alguna **clase**.

- Una clase es un objeto que abstrae el comportamiento común de todas sus instancias.
  Ejemplo. `(1 @ 2)` es una instancia de la clase `Point`.
- Todas las instancias de una clase tienen los mismos atributos.
  Ejemplo. Todas las instancias de `Point` tienen atributos `x` e `y`.
- Todas las instancias de una clase usan el mismo método para responder un mismo mensaje.
  Ejemplo. Los mensajes `(1 @ 2) rho` y `(3 @ 4) rho` se resuelven con un método implementado en la clase `Point`.

  Veremos más sobre *method dispatch* más adelante.

Subclasificación y herencia

Cada clase es **subclase** de alguna otra clase.
Las clases se estructuran en una jerarquía. Por ejemplo:



Una clase **hereda** todos los métodos de su superclase.

Una clase puede elegir **reemplazar**[4] un método definido en la superclase por otro más específico.

Hay clases que están destinadas a abstraer el comportamiento de sus subclases, pero no tienen instancias (p. ej. la clase `Number`). Estas se llaman clases **abstractas**.

[4] En inglés, *override*.

Introducción

Conceptos fundamentales de POO

Introducción a la POO en Smalltalk

Ejemplos básicos

Evaluemos los siguientes comandos:

1. `1 + 2`
2. `1 + 2 * 3` (Cuidado con la precedencia)
3. `1 class`
4. `1 class superclass`
5. `3 squared / 2 squared`
6. `a := Array new: 10`
7. `a at: 1 put: 'hola'`
8. `a at: 1`

(Juguemos un poco con el entorno)

Definamos una clase `Par` y los siguientes métodos:

1. `Par x: unObjeto y: otroObjeto` — construye un par.
2. `par x`, `par y` — proyectan las componentes.
3. `par + otroPar` — suma dos pares
4. ¿Qué sucede si anidamos pares y usamos la suma?

Sintaxis de expresiones y comandos

Smalltalk es un **entorno orientado a objetos**.
No es tan apropiado pensarlo como un *lenguaje*.
Pero necesitamos una sintaxis concreta para describir los métodos.

Sintaxis "mínima" de expresiones y comandos

Expr	::=	`x`	*nombre local*
	|	`X`	*nombre global*
	|	*Expr* `msg`	*mensaje unario*
	|	*Expr* ⟨`op`⟩ *Expr*	*mensaje binario*
	|	*Expr* `msg1:` $Expr_1$ `...msgN:` $Expr_N$	*mensaje keyword*
	|	`x :=` *Expr*	*asignación*
Cmd	::=	*Expr*	*expresión*
	|	^*Expr*	*retorno*
	|	*Expr*. *Cmd*	*secuencia*

Los nombres locales se refieren a variables, atributos y parámetros.
Precedencia: mensajes unarios $>$ binarios $>$ keyword.

Sintaxis de Smalltalk

Algunos otros elementos sintácticos:

- Las variables locales se declaran con `|x1 ... xn|`.
- Los mensajes se pueden encadenar con “;”.
- Hay seis palabras reservadas:

  `nil` `true` `false` `self` `super` `thisContext`

- Se incluye notación para diversos tipos de literales:
  - Constantes numéricas: `29`, `-1.5`, ...
  - Caracteres: `$a`, `$b`, `$c`, ...
  - Símbolos: `#hola`, ...
  - Strings: `'hola'`, ...
  - ...

Polimorfismo

Una misma operación puede operar con objetos que implementan la misma interfaz de distinta manera.

Esta característica de la POO se puede aprovechar para construir programas genéricos, que operan con objetos independientemente de sus características específicas.

Ejemplo

```
z := OrderedCollection new.
z add: Gato new; add: Perro new; add: Pato new.
z do: [:animal | animal hablar].   "miau guau cuac"
```

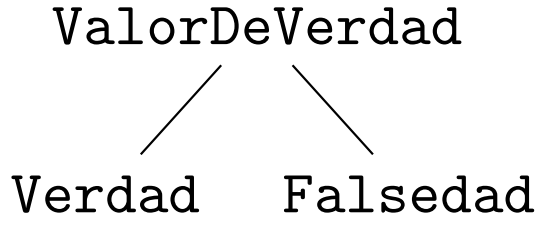
- No es necesario hacer un `switch/case` para analizar de qué especie de animal se trata.
- Cada objeto implementa su propio método para responder al mensaje "`hablar`" de forma correspondiente.

Polimorfismo y estructuras de control

Smalltalk no tiene estructuras de control (`if`, `for`, `while`, etc.).
Estos comportamientos se implementan con envíos de mensajes.

Veamos cómo implementar manualmente un condicional.

1. Definamos la siguiente jerarquía de clases:



2. Definamos métodos para implementar el mensaje:

```
unValorDeVerdad siEsVerdadero: x siEsFalso: y
```

3. Definamos métodos para implementar el mensaje:

```
unValorDeVerdad not
```

   aprovechando el polimorfismo.

4. ¿Qué sucede si evaluamos el siguiente comando?

```
Verdad new siEsVerdadero: 1
           siEsFalso: Misil new lanzar
```

Bloques

Un *bloque* o *clausura* es un objeto que representa un comando, es decir una secuencia de envíos de mensajes.

Se extiende la sintaxis:

Expr	::=	...	
	|	`[`*Cmd*`]`	*bloque sin parámetros*
	|	`[:x1 ... :xN |`*Cmd*`]`	*bloque con parámetros*

Los bloques sin parámetros se pueden evaluar con el mensaje:

```
bloque value
```

Los bloques con *N* parámetros se pueden evaluar con el mensaje:

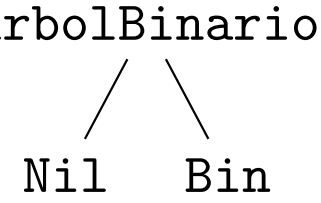
```
bloque value: arg1 value: arg2 ... value: argN
```

Ejemplo

1. `[1 + 1] value`
2. `[:x :y | x + y] value: 1 value: 2`
3. Modifiquemos el condicional para que opere con bloques.

Ejemplo — recorrido sobre árboles

1. Definamos la siguiente jerarquía de clases:



   y los siguientes métodos, con la semántica esperable:

   ```
   Nil new
   Bin izq: unArbol raiz: unDato der: otroArbol
   ```

2. Definamos métodos para implementar el siguiente mensaje, que recibe un bloque de un parámetro y lo ejecuta sobre todos los elementos del árbol siguiendo el recorrido inorder:

   ```
   arbol do: bloque
   ```

3. Definamos un método para implementar el mensaje:

   `arbol size` (implementarlo usando `do:`)

Dos puntos interesantes:

- Los bloques pueden mutar el estado de variables capturadas.
- El método `size` funciona para cualquier colección que acepte el mensaje `do:`.

Manejo de mensajes incomprendidos

¿Qué sucede si evaluamos el siguiente comando?

```
10 contarHasta: 20
```

- El objeto `10` recibe un mensaje `doesNotUnderstand: msg`.
- `msg` es una instancia de la clase `Message` que *reifica* el mensaje "contarHasta: 20".

Herencia y reutilización de código

Definamos una clase `Robot` con la siguiente interfaz:

1. `robot initialize` — lo inicializa en la posición `0@0`.
2. `robot posicion` — devuelve la posición actual.
3. `robot desplazar: vector` — modifica la posición actual sumándole un vector (que acepta mensajes `x` e `y`).

Definamos ahora una subclase `RobotConUndo`:

1. `robot undo` — deshace el último desplazamiento[5].

Nos vemos obligados a reemplazar[6] los métodos `initialize` y `posicion`. Pero no queremos copiar y pegar el código.

Super

La palabra reservada `super` se refiere al mismo objeto que `self`. Pero `super m` indica que la búsqueda del método que implementa el mensaje `m` debe comenzar desde la superclase de la clase actual.

[5] Podemos usar `OrderedCollection`, `add:` y `removeLast`.

[6] *Override*.

Algoritmo de resolución de métodos (*method dispatch*)

Entrada

O: objeto al que se le desea enviar un mensaje.

S: selector del mensaje que se desea enviar. (P.ej.: `at:put:`)

C: clase en la que se desea buscar el método.

Salida

M: método que se debe ejecutar para responder el mensaje,

o `NotUnderstood` en caso de que no exista.

Procedimiento

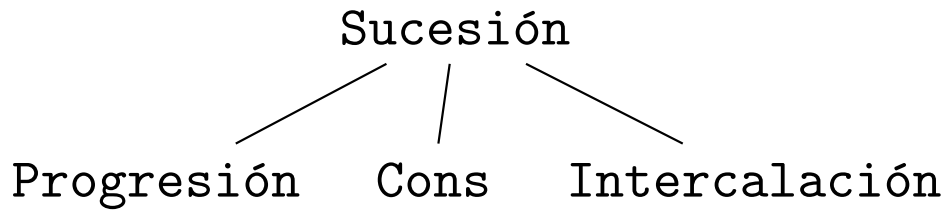
1. Si C define un método M para S, devolver M.
2. Si no, sea C' la superclase de C.
   - 2.1 Si C' es `nil`, devolver `NotUnderstood`.
   - 2.2 Si no, asignar $C := C'$ y volver al paso 1.

Algoritmo de resolución de métodos (*method dispatch*)

- En general, cuando se envía un mensaje, se usa el algoritmo de resolución de métodos tomando como C la clase del objeto O.
- **Caso particular:**
  cuando se envía un mensaje a `self`, O es el mismo objeto receptor.
- **Excepción:**
  cuando se envía un mensaje a `super`, O es el mismo objeto receptor mientras que C es la superclase de la clase del método que contiene el envío de mensaje a `super`.

Ejercicio — Streams

Un *stream* es un objeto que representa una sucesión infinita. Acepta un mensaje `prox`, que devuelve el elemento actual y avanza al siguiente elemento. Definimos una jerarquía de clases:



1. `Progresión desde: x aplicando: bloque` — construye un *stream* que tiene a `x` como primer elemento y calcula el siguiente elemento usando el bloque.
2. `Cons cabeza: unElemento cola: unStream` — extiende al *stream* dado con un elemento en la cabeza.
3. `Intercalación entre: s1 y: s2` — construye un *stream* que alterna entre los elementos de `s1` y los de `s2`.
4. (Más difícil). Definir un método para implementar el mensaje `dividir`, que devuelve dos *streams* que intercalados resultarían en el *stream* original.

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿? ? ? ? ? ? ? ? ?